The Gaming Blog Application Project Documentation

By: Alexis Pechon

# Introduction

The purpose of this project was to develop a Gaming Blog that allows users to sign up or sign into their accounts while being able to enter, delete, edit, or update their gaming entries on the blog entry that they have made. Likewise, the users should be able to leave a comment on the website and be able to delete or edit those comments respectively. This project must be built using Ruby on Rails while also implementing External APIs, External Gems, Design Patterns, and a user-friendly GUI as well as having the basic CRUD functionality that allows the user to add, read, update, and delete their contributions to the blog whenever it is appropriate or necessary.

**Use Case Diagram**

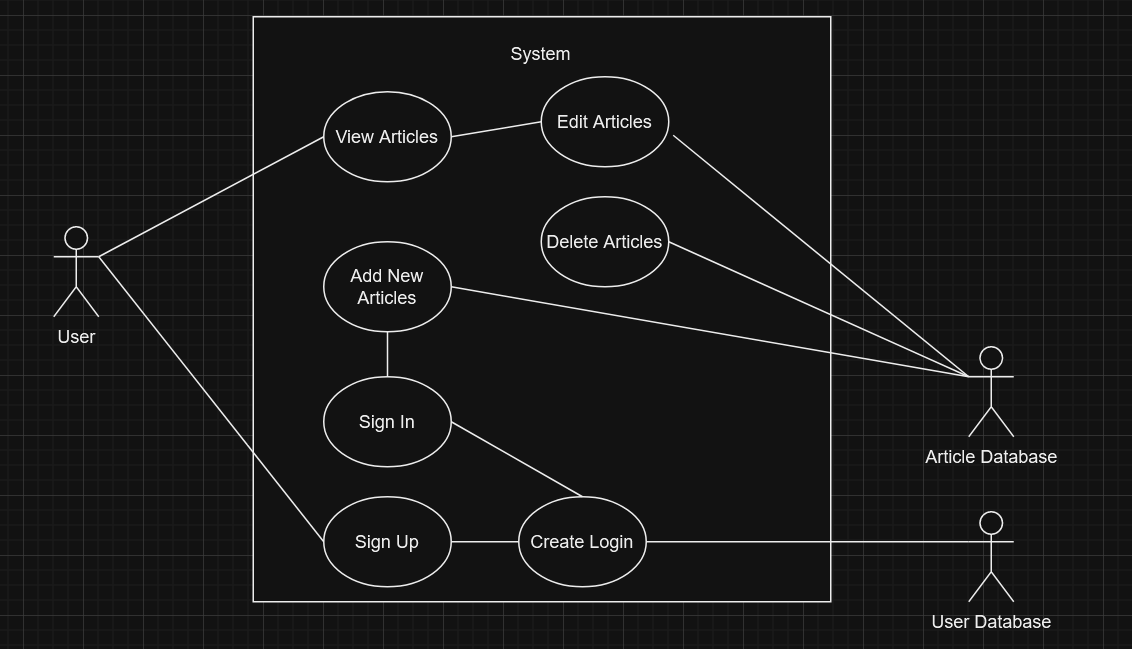


Figure 1: The screenshot shows the Use Case Diagram explaining how the major components of my project interact with each other, such as the User and the Databases.

# Development Strategy

## Database Design

Before developing the application, one of the biggest concerns was the database. Since the project heavily relies on storing information from the user and displaying it on the webpage, a database is mandatory, otherwise, the information could never be displayed.

To counter this project, I have developed two different databases that both contribute to the overall functionality of the webpage. The first database would be responsible for holding the id, title, and the main body of the games that the user would post on the blog.

To create this database, a command had to be executed using the Command Line Interface or the CMD. This must be done in the same folder in which the Ruby on Rails Project is kept, which in this case would be the “blog” folder. After opening the CMD, I ran the command: “**rails generate model Article id:integer title:string body:string**”. After running the command, it should generate the basic structure for the Article Database. Afterward, I needed to execute the command “rails db:migrate” to send the information to the database.

This was a similar strategy I used for creating the User model required for the User Login Functionality of the project. The command that I used was: “**rails generate model User email:string password:string**”, and it generated the model in which the basic user login functionality could be integrated into my project. The Article and User model both share a 1:1 Relationship as the User is needed for user to upload their games onto the blog and the Article model is needed to store the different contributions and entries that the user has made to the blog.

## Implementation

**Functionality:** Create a Gaming Article – The user should be able to contribute to the blog by being able to create, read, update, and delete an article from the blog once they have set up their account through the user login functionality.

The Gaming Blog Application was developed using the MVC, or the Model-View-Controller, approach.

**Model –** The file “**article.rb”** contains the logic that is required for people to add their games to the blog with a few constraints.

**View –** The file “**index.html.erb”** is the most important file in the application as it controls all the HTML Elements and logic that are required to display the contents of the webpage to the user when they first visit the website. Furthermore, this is an important file as it also contains the code that links the index page to the user login and account creation page, allowing the user to create or sign into their accounts respectively.

**Controller –** The file “**articles\_controller.rb**” is the core structure of the blog application as the file itself controls the different methods that allow the user to create, read, update, and delete their entries to the blog whenever it is necessary. Without this file then most of the website’s functionality would be lost and wouldn’t be able to operate as intended.

**Functionality –** Creating a Discord Widget using Discord’s External API.

**Implementation –** The way I implemented this feature into my project is that I took this code from the Widget menu on Discord. The server widget must be enabled for this to work. Once that is completed, I took the JSON API link: <https://discord.com/api/guilds/973969434127507526/widget.json> and what it does is that it grabs the information of all the different server members that are on a Discord Server. This information will be displayed on the screen with the use of the “fetch\_data.html.erb” file which takes specific ids from the API and formats it so that the data can be easily displayed for the user. Likewise, the “discord\_widget\_controller.html.erb” is essential as it contains the logic to carry out this particular functionality of the project.

**Functionality –** Implementing a Dad Joke to be shown at the bottom of the webpage using the Dad Joke External API.

**Implementation –** The way that the Dad Joke API was implemented into my project was by using this link: “https://v2.jokeapi.dev/joke/Programming?blacklistFlags=racist,sexist,explicit” and developing the code inside of the controllers folder inside of the “joke\_controller.rb” file. The way this would be later displayed to the user, meaning that they could access it through a link on the main page of the application and bet sent to a page where the joke is shown. Once implemented, this would be output that the user would be able to see when they visit the webpage.

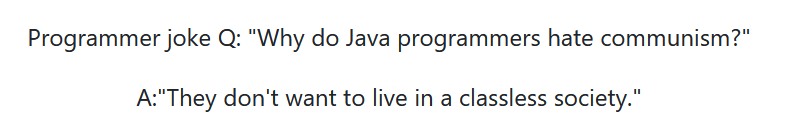


Figure 2: The screenshot above shows the Dad Joke API that is implemented into my project working and displaying the contents of the API.

## Design Patterns

The Design Patterns were a crucial part of the project’s development as they also contributed to the project’s overall functionality and completeness. The two design patterns that I have decided to use were the Singleton and Observer Design Patterns.

**Singleton Design Pattern –** The first Design Pattern that was implemented into my project was the Singleton Design Pattern. The main reason why I decided to implement this design pattern is that the Singleton Design Pattern restricts the initialization of the class so that only one of the classes can be created and used throughout the program. This is especially useful for a blog since I want to have a single instance of the project running for all clients, so everyone can view the same information as displayed on the screen. In my project, the Singleton Design Pattern was best utilized inside of the articles\_controller.rb file inside of the create method. This means only one instance of that class could be executed which is useful as it also tracks the title of the entries of the games that were being submitted to the blog.

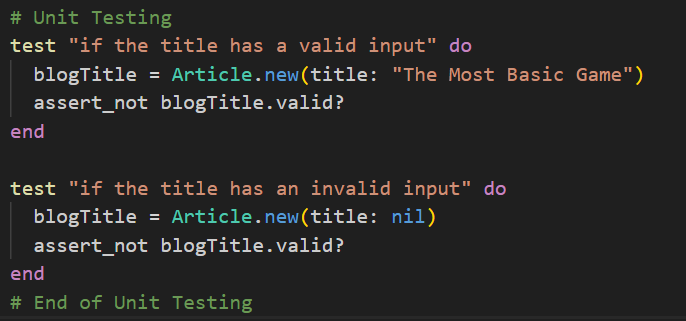
**Observer Design Patterns –** The Observer Design Pattern was the second Design Pattern that was implemented into my project. The main reason why I chose this design pattern was that I wanted to have a feature in my program that tracks the changes made to the blog, such as changes made to the entries’ descriptions or the entries’ deletions. By implementing the observer design pattern into my project, I was able to monitor the different changes being made to my application. This feature was also implemented in the same “articles\_controller.rb” file as the Singleton Design Pattern.

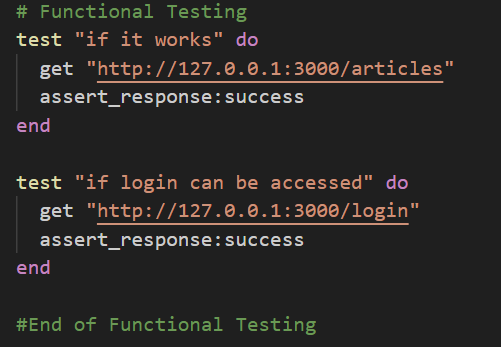
# Testing

This section should describe how you followed a Test-Driven Development approach in the creation of your application. You should include details on how you have implemented Unit, Functional, and Integration testing along with suitable examples.

Test-Driven Development played a crucial role in the development of this project. The practice of Test-Driven Development was considered since the initial development of my Ruby on Rails Project. How this was done was that after creating the project, models, and controllers using the command line interface or the CMD, Ruby on Rails would then generate a special “test” folder where all the different test cases could be implemented in.

Before I could even start the project, I had to develop the different test cases that were going to be executed to test the different core functionalities of my project. Using the testing folder that was generated by Ruby on Rails, I began creating the different test cases in the “articles\_controller\_test.rb” inside of the controllers folder. This is where I implemented Unit, Functional, and Integration Testing.





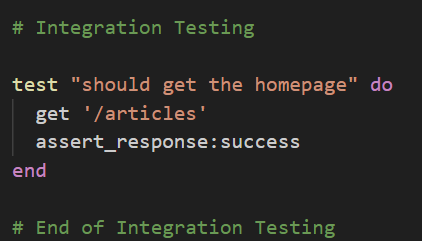


Figure 3: The previous 3 screenshots that were provided are all examples of code snippets of the different test cases that were created as part of the Test-Driven Requirement.

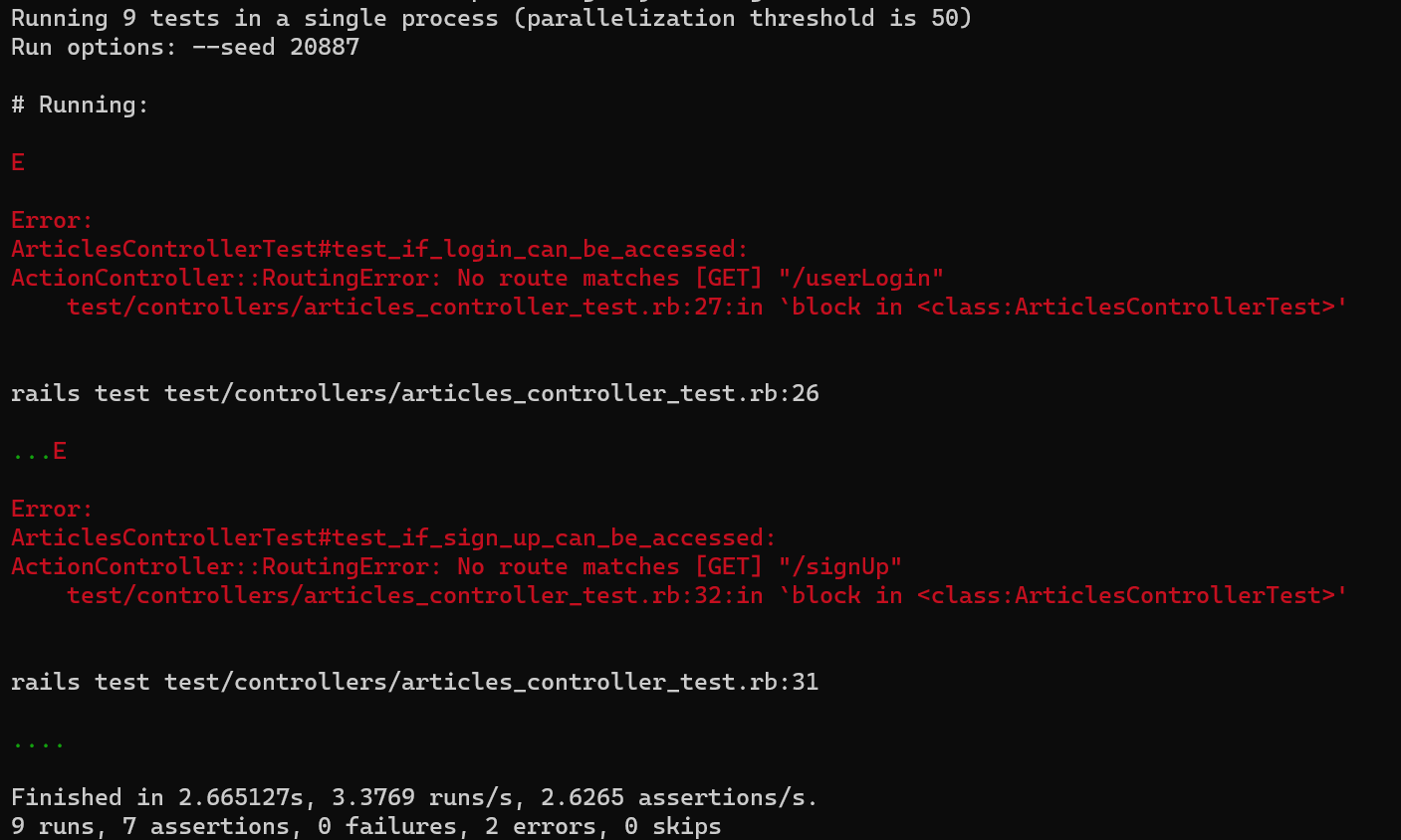


Figure 4: The screenshot shown above is an example of the Tests I ran using the CMD based on the Test Cases that I previously created at the start of the project’s development.

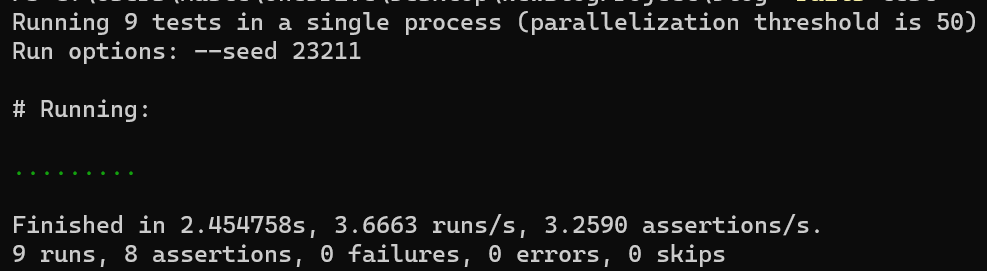


Figure 5: The screenshot above is an example of the Tests I ran using CMD, however, these tests were executed nearing towards the end of the project’s development once all the features were correctly implemented.

# Deployment, Continuous Integration, and Delivery strategies

**AWS Lambda**

AWS Lambda is an example of a computer service that enables programmers to execute and run their code without any provisioning as well as server management. How Lambda works is that it will execute the code on a high-availability computer infrastructure while simultaneously performing all the administration of the computing resources for the servers and operating systems, as well as capacity provisioning and automatic scaling. Furthermore, programmers could also organize their code to include Lambda functions, meaning that Lambda will only run the functions when it is needed.

**Advantages –** In a scenario where scalability is a mandatory requirement, AWS Lambda is the ideal computing service. This is because AWS Lambda offers users a wide variety of services, some examples would include the IoT backend, which is useful for integrating third-party API requests, streamlined processing as well as file processing which enables real-time data processing after an upload.

**Disadvantages –** Some of the limitations, or disadvantages of using AWS Lambda, are its 512MB Disk Space, the request and response body payload can be up to 6MB in size, and the deployment package is limited to 50MB in size. The reason for the 50MB limit is that it discourages users to directly uploading their deployment package directly onto Lambda.

**Microsoft Azure**

Microsoft Azure is another example of a Cloud Computing platform. Microsoft Azure provides a wide variety of cloud services, some of which include computing, analytics, storage as well as networking. Depending on the scope of the project, users could easily choose from those different options to develop or scale new applications while also being capable of running existing applications on the cloud.

**Advantages –** There are multiple different advantages related to Microsoft Azure, some include Network Security, Scalability as well High Availability. In the context of High Availability, this is a tremendous benefit for programmers whose projects must remain up and functioning with little to no downtime. This is because Microsoft Azure has access to different data centers on a global scale, and due to that global reach, this means that the downtime for Microsoft Azure can be as low as 5 hours annually.

**Disadvantages –** However, there are also still some drawbacks when using Microsoft Azure. The first drawback is the necessary expertise required to be able to work properly, making it less than ideal for inexperienced programmers to work with due to the lack of experience to correctly and efficiently run their application on the platform. Furthermore, along with needing an experienced programmer to work with Azure, there should also be a dedicated professional who will be able to manage the data since Microsoft Azure doesn’t provide any assistance when it involves data center management.

**Google App Engine (GAE)**

Similarly, to Microsoft Azure, Google App Engine, otherwise known as GAE, is a Platform as a Service that enables developers to have access to Google’s scalable hosting as well as 1-tier internet service while being able to deploy their application. However, to fully utilize GAE, the application must be written or programmed using Java or Python while also storing data in Google’s Bigtable and must use Google’s query language. In other words, any application or project that fails to meet these rules and restrictions will not be able to work correctly.

**Advantages –** Just like with other Platforms as a Service, there are various advantages and disadvantages associated with them. For the Google App Engine as a platform, there are various advantages that they can provide to programmers, some examples would include Easy Setup and Use. Due to GAE being fully managed, programmers can create the code without the assistance of IT Operations and back-end infrastructures. With that in mind, GAE also provides programmers with different built-in APIs that could be used in the development of different projects and applications. Another advantage would be scalability, which means that the GAE engine scales with the workload as contents are being added and removed while managing the application’s resources.

**Disadvantages –** However, GAE, much like other Platforms as a Service, also comes with serval drawbacks that may need to be considered when choosing GAE as a platform. The first drawback is the performance issues associated with GAE. If an application or project heavily relies on CPU Usage, this could potentially slow down operations on GAE while also being more expensive due to the resources that are required to run the application. Another disadvantage is Limited Access. When using GAE, developers will only have limited access to the files, this is largely because files inside of GAE have read-only access, meaning that it can’t be edited or changed later. Lastly, another disadvantage of using GAE also includes a Lack of Control, which means that if something were to happen to GAE as a platform, then the programmer is heavily reliant on Google themselves to resolve the issue, which could also potentially lead to longer or unintentional downtimes for their applications.

**Implementing Continuous Integration and Delivery using CircleCI**

Continuous Integration and Continuous Development, also commonly referred to as CI/CD, is one of the processes that I had to consider throughout the development of the project. The main reason for this was to keep a steady workflow where the application could be deployed as new features and code are being added to the application in parallel with each other. Furthermore, in the scope of my project, I met the Continuous Integration and Continuous Delivery strategies by creating a CICD Pipeline with the use of CircleCI. With all that put into consideration, I was able to efficiently and continuously develop my application while simultaneously observing the CI/CD Pipeline logs for any errors or noticeable changes in the application.

The way this was integrated into my blog project was by creating a folder inside of the blog folder titled “.circleci”, in which a file titled “config.yml” could be found. This is essential for setting up the CI/CD Process as these were the major prerequisites that were required by CircleCI to correctly establish the Pipeline and start the CI/CD Process. Likewise, the blog project must be stored in a GitHub repository as CircleCI needs to have access to the project itself. Once CircleCI is synced up with my GitHub Repository I was able to configure and set up the CI/CD Pipeline using the .circleci folder and config.yml files.

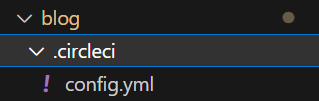


Figure 6: The screenshot shows the contents of the .circleci folder with the config.yml file inside of it. This is necessary for starting the CI/CD Process using CircleCI.

A screenshot of a computer

Description automatically generated

Figure 7: The screenshot shows my Ruby on Rails Project running on CircleCI once all of the different configurations and setup were completed.

Link to the CircleCI Pipeline: <https://app.circleci.com/pipelines/github/AlexisPechon>